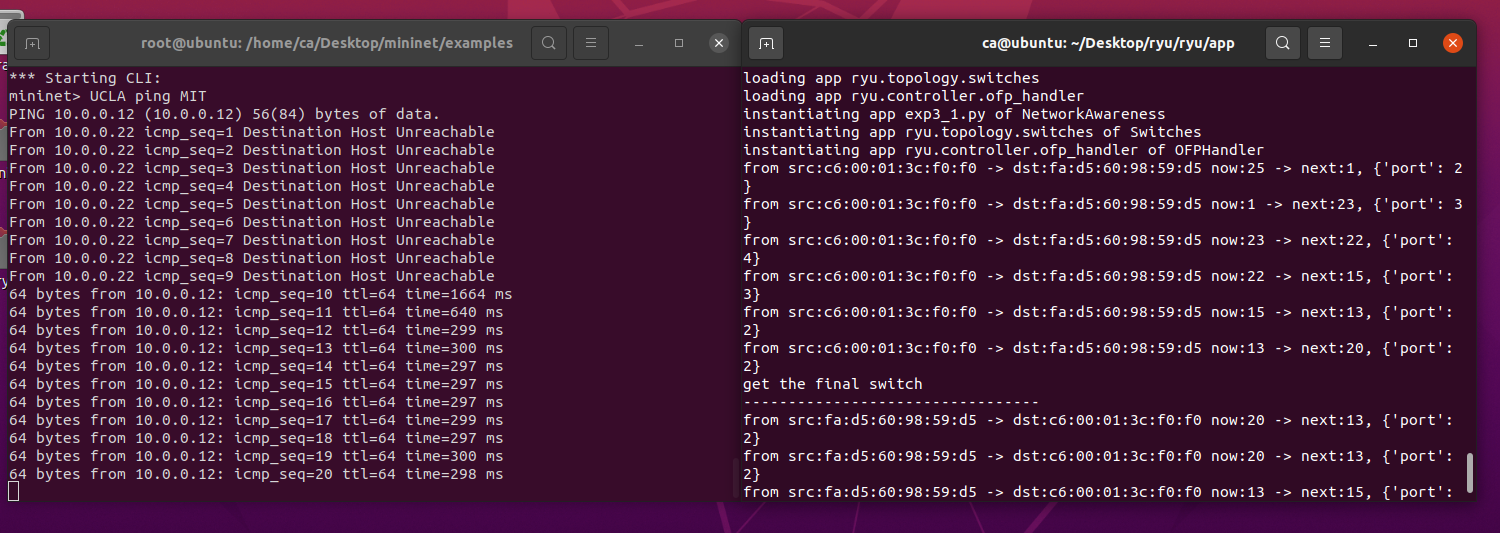
西安交通大学SDN第三次实验

Exp1:

在mininet中执行命令: python Arpanet19723.py --controller remote

在ryu中执行命令: ryu-manager exp3\_1.py --observe-links

结果如图:



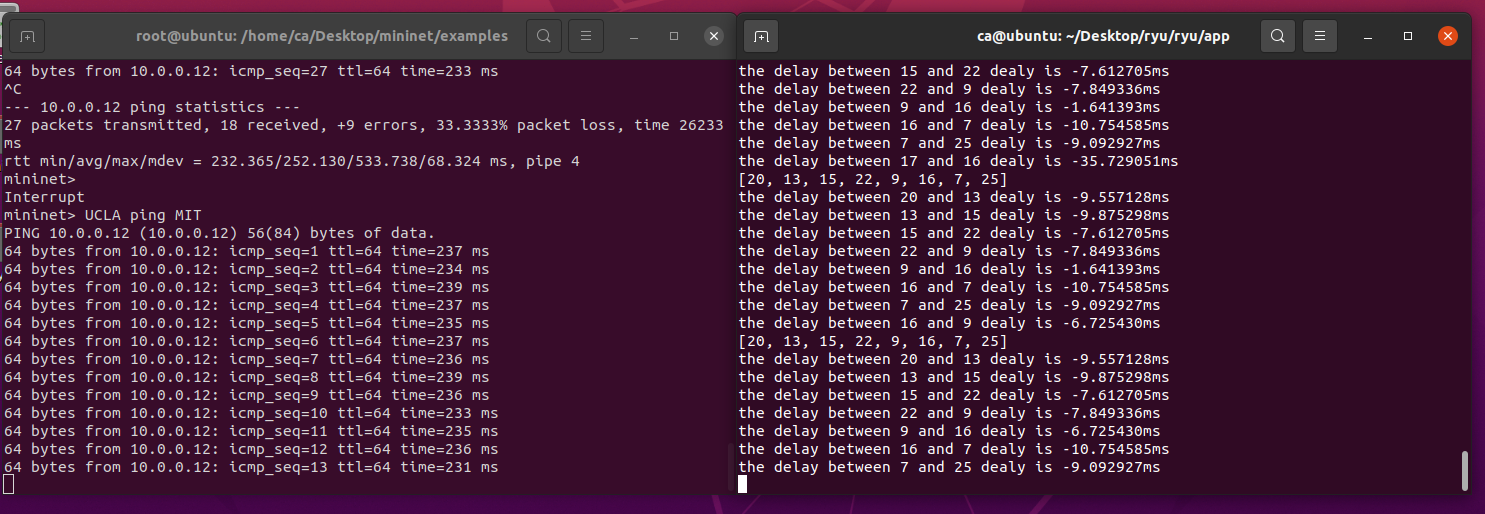
可以看到, 最小跳数为 25->1->23->22->15->13->20

EXP2:

在mininet中执行命令: python Arpanet19723.py --controller remote

在ryu中执行命令: ryu-manager exp3\_2.py --observe-links

结果如图:



可以看到, 最小时延的路径为: 20->13->15->22->9->16->7->25

整体思路:

Exp1中, 当 控 制 器 收 到 packetin 包 的 时 候 ,调 用

packet\_in\_handler()函数，如果是 lldp 包或者是 ipv6 包则 return。

如果是 arp 包，则将其 ip 地址、交换机、源地址和端口相对应（学

习过程）。根据该包内的 mac 地址确定其传播方式（和第一个实验相

同）。然后根据所给的拓扑，依赖 networkx 构建相应的拓扑图，然后

调用 api 算出最短路径。（其中每条路径的权值都是 1）

a.在初始化的时候，添加以下列表：

self.mac\_table = {}

#学习 mac 地址，确定输出端口

self.arp\_anti\_loop = {} #用于打破环路

self.arp\_table = {}

self.topo\_thread = hub.spawn(self.\_get\_topology)

self.graph = nx.DiGraph()

#创建一个图利用

networkx 自带 api 计算最短路径

b.过滤 lldp 包和 ipv6 包（ipv6 不使用 arp 协议）

# ignore lldp packet

if eth\_pkt.ethertype == ether\_types.ETH\_TYPE\_LLDP:

return

if eth\_pkt.ethertype == ether\_types.ETH\_TYPE\_IPV6:return

c.获取 header list（进而获取 ip 地址）

header\_list

=

dict((p.protocol\_name,

p)for

p

in

pkt.protocols if type(p) != str)

d.根据 arp 包学习的过程，打破环路

if dst == ETHERNET\_MULTICAST and ARP in header\_list:

#如

果是 arp 包

arp\_dst\_ip = header\_list[ARP].dst\_ip

if (dpid, src, arp\_dst\_ip) in self.arp\_anti\_loop:

if

self.arp\_anti\_loop[(dpid,

src,

arp\_dst\_ip)] != in\_port:

out = parser.OFPPacketOut(

datapath=dp,

buffer\_id=ofp.OFP\_NO\_BUFFER,

in\_port=in\_port,

actions=[], data=None)

dp.send\_msg(out)

return

else:

self.arp\_anti\_loop[(dpid, src, arp\_dst\_ip)]

= in\_porte.获取输出端口

if self.mac\_table[dpid].has\_key(dst):

out\_port = self.mac\_table[dpid][dst]

else:

out\_port = ofp.OFPP\_FLOOD

Exp2中, a.初始化时候，添加如下列表

self.mac\_to\_port = {}

self.network = nx.DiGraph()

self.graph = nx.DiGraph()

self.paths = {}

self.topology\_api\_app=self

self.echo\_latency={}

self.request\_latency={}

self.datapaths={}

self.sw\_module = lookup\_service\_brick('switches')

self.awareness = lookup\_service\_brick('awareness')

self.network\_aware

=

lookup\_service\_brick('network\_aware')b.学习 mac 地址的函数

mac\_learning() #与问题一相同，不在赘

述

def mac\_learning(self, datapath, src, in\_port):

self.mac\_to\_port.setdefault((datapath,datapath.id),

{})

# learn a mac address to avoid FLOOD next time.

if src in self.mac\_to\_port[(datapath,datapath.id)]:

if

in\_port

!=

self.mac\_to\_port[(datapath,datapath.id)][src]:

return False

else:

self.mac\_to\_port[(datapath,datapath.id)][src] =

in\_port

return True

c.计算延时的函数

def get\_delay(self, src, dst):

try:

fwd\_delay = self.request\_latency[(src,dst)]

re\_delay = self.request\_latency[(dst,src)]

src\_latency = self.echo\_latency[src]

dst\_latency = self.echo\_latency[dst]

delay = (fwd\_delay + re\_delay - src\_latency -dst\_latency)\*(1000/2)

print('the

delay

between

%s

and

%s

dealy

is %fms'%(src,dst,delay))

return max(delay, 0)

except:

print("get delay error")

return float('inf')

d.获取 lldp 的延时，然后赋值给相应拓扑图中的路径，作为其权值。

if eth.ethertype == ether\_types.ETH\_TYPE\_LLDP:

try:

src\_dpid,

src\_port\_no

=

LLDPPacket.lldp\_parse(msg.data)

dpid = datapath.id

if self.sw\_module is None:

self.sw\_module

=

lookup\_service\_brick('switches')

if src\_dpid not in self.paths.keys():

self.paths.setdefault(src\_dpid, {})

for port in self.sw\_module.ports.keys():

if src\_dpid == port.dpid and src\_port\_no

== port.port\_no:

port\_data

=self.sw\_module.ports[port]

delay = port\_data.delay

self.request\_latency[(src\_dpid,dpid)] = delay

# print('lldp delay between %s and %s

is %fms'%(src\_dpid,dpid,delay\*1000))

self.network[src\_dpid][dpid]['weight']

= self.get\_delay(src\_dpid, dpid)

if dpid in self.network:

if

dpid

not

in

self.paths[src\_dpid]:

path

=

nx.shortest\_path(self.network,src\_dpid,dpid)

self.paths[src\_dpid][dpid]=path

path = nx.shortest\_path(self.network,20,25,

weight='weight')

print(path)

total = 0

for i in range(len(path)-1):

total += self.get\_delay(path[i], path[i+1])

# print("total:", total)except Exception as e:

print(e)

print("error occured")

finally:

return

e.确定其输出端口之后基本上与问题一相同，不在赘述，包含建立

topo 和计算最短路径（带权值）

if dst in self.mac\_to\_port[(datapath,datapath.id)]:

out\_port

=

self.mac\_to\_port[(datapath,datapath.id)][dst]

else:

if self.mac\_learning(datapath, src, in\_port) is

False:

out\_port = ofproto.OFPPC\_NO\_RECV

else:

out\_port = ofproto.OFPP\_FLOOD

源码:

Arpanet19723:

#!/usr/bin/python

"""

Custom topology for Mininet, generated by GraphML-Topo-to-Mininet-Network-Generator.

"""

from mininet.topo import Topo

from mininet.net import Mininet

from mininet.node import RemoteController

from mininet.node import Node

from mininet.node import CPULimitedHost

from mininet.link import TCLink

from mininet.cli import CLI

from mininet.log import setLogLevel

from mininet.util import dumpNodeConnections

class GeneratedTopo( Topo ):

"Internet Topology Zoo Specimen."

def \_\_init\_\_( self, \*\*opts ):

"Create a topology."

# Initialize Topology

Topo.\_\_init\_\_( self, \*\*opts )

# add nodes, switches first...

s1 = self.addSwitch( 's1' )

s2 = self.addSwitch( 's2' )

s3 = self.addSwitch( 's3' )

s4 = self.addSwitch( 's4' )

s5 = self.addSwitch( 's5' )

s6 = self.addSwitch( 's6' )

s7 = self.addSwitch( 's7' )

s8 = self.addSwitch( 's8' )

s9 = self.addSwitch( 's9' )

s10 = self.addSwitch( 's10' )

s11 = self.addSwitch( 's11' )

s12 = self.addSwitch( 's12' )

s13 = self.addSwitch( 's13' )

s14 = self.addSwitch( 's14' )

s15 = self.addSwitch( 's15' )

s16 = self.addSwitch( 's16' )

s17 = self.addSwitch( 's17' )

s18 = self.addSwitch( 's18' )

s19 = self.addSwitch( 's19' )

s20 = self.addSwitch( 's20' )

s21 = self.addSwitch( 's21' )

s22 = self.addSwitch( 's22' )

s23 = self.addSwitch( 's23' )

s24 = self.addSwitch( 's24' )

s25 = self.addSwitch( 's25' )

# ... and now hosts

h1 = self.addHost( 'ILLINOIS' )

h2 = self.addHost( 'MITRE' )

h3 = self.addHost( 'CARNEGIE' )

h4 = self.addHost( 'CASE' )

h5 = self.addHost( 'ETAC' )

h6 = self.addHost( 'AFGWC' )

h7 = self.addHost( 'BBN' )

h8 = self.addHost( 'NBS' )

h9 = self.addHost( 'Tinker' )

h10 = self.addHost( 'AMES' )

h11 = self.addHost( 'RADC' )

h12 = self.addHost( 'McClellan' )

h13 = self.addHost( 'RAND' )

h14 = self.addHost( 'AMES13' )

h15 = self.addHost( 'SDC' )

h16 = self.addHost( 'BBN15' )

h17 = self.addHost( 'HARVARD' )

h18 = self.addHost( 'SRI' )

h19 = self.addHost( 'UCSB' )

h20 = self.addHost( 'UCLA' )

h21 = self.addHost( 'Stanford' )

h22 = self.addHost( 'USC' )

h23 = self.addHost( 'UTAH' )

h24 = self.addHost( 'Lincoln' )

h25 = self.addHost( 'MIT' )

# add edges between switch and corresponding host

self.addLink( s1 , h1 )

self.addLink( s2 , h2 )

self.addLink( s3 , h3 )

self.addLink( s4 , h4 )

self.addLink( s5 , h5 )

self.addLink( s6 , h6 )

self.addLink( s7 , h7 )

self.addLink( s8 , h8 )

self.addLink( s9 , h9 )

self.addLink( s10 , h10 )

self.addLink( s11 , h11 )

self.addLink( s12 , h12 )

self.addLink( s13 , h13 )

self.addLink( s14 , h14 )

self.addLink( s15 , h15 )

self.addLink( s16 , h16 )

self.addLink( s17 , h17 )

self.addLink( s18 , h18 )

self.addLink( s19 , h19 )

self.addLink( s20 , h20 )

self.addLink( s21 , h21 )

self.addLink( s22 , h22 )

self.addLink( s23 , h23 )

self.addLink( s24 , h24 )

self.addLink( s25 , h25 )

# add edges between switches

self.addLink( s1 , s25, bw=10, delay='50ms')

self.addLink( s1 , s23, bw=10, delay='34ms')

self.addLink( s2 , s3, bw=10, delay='13ms')

self.addLink( s2 , s5, bw=10, delay='14ms')

self.addLink( s3 , s4, bw=10, delay='15ms')

self.addLink( s4 , s11, bw=10, delay='12ms')

self.addLink( s4 , s6, bw=10, delay='17ms')

self.addLink( s5 , s8, bw=10, delay='10ms')

self.addLink( s7 , s25, bw=10, delay='18ms')

self.addLink( s7 , s16, bw=10, delay='17ms')

self.addLink( s8 , s17, bw=10, delay='13ms')

self.addLink( s9 , s22, bw=10, delay='14ms')

self.addLink( s9 , s16, bw=10, delay='19ms')

self.addLink( s10 , s18, bw=10, delay='14ms')

self.addLink( s10 , s14, bw=10, delay='15ms')

self.addLink( s11 , s24, bw=10, delay='17ms')

self.addLink( s12 , s18, bw=10, delay='40ms')

self.addLink( s12 , s23, bw=10, delay='44ms')

self.addLink( s13 , s20, bw=10, delay='15ms')

self.addLink( s13 , s21, bw=10, delay='18ms')

self.addLink( s13 , s15, bw=10, delay='15ms')

self.addLink( s14 , s21, bw=10, delay='19ms')

self.addLink( s15 , s22, bw=10, delay='15ms')

self.addLink( s16 , s17, bw=10, delay='12ms')

self.addLink( s18 , s19, bw=10, delay='44ms')

self.addLink( s19 , s20, bw=10, delay='48ms')

self.addLink( s22 , s23, bw=10, delay='16ms')

self.addLink( s24 , s25, bw=10, delay='13ms')

topos = { 'generated': ( lambda: GeneratedTopo() ) }

# HERE THE CODE DEFINITION OF THE TOPOLOGY ENDS

# the following code produces an executable script working with a remote controller

# and providing ssh access to the the mininet hosts from within the ubuntu vm

controller\_ip = ''

def setupNetwork(controller\_ip):

"Create network and run simple performance test"

# check if remote controller's ip was set

# else set it to localhost

topo = GeneratedTopo()

if controller\_ip == '':

#controller\_ip = '10.0.2.2';

controller\_ip = '127.0.0.1';

net = Mininet(topo=topo, controller=lambda a: RemoteController( a, ip=controller\_ip, port=6633 ), host=CPULimitedHost, link=TCLink)

return net

def connectToRootNS( network, switch, ip, prefixLen, routes ):

"Connect hosts to root namespace via switch. Starts network."

"network: Mininet() network object"

"switch: switch to connect to root namespace"

"ip: IP address for root namespace node"

"prefixLen: IP address prefix length (e.g. 8, 16, 24)"

"routes: host networks to route to"

# Create a node in root namespace and link to switch 0

root = Node( 'root', inNamespace=False )

intf = TCLink( root, switch ).intf1

root.setIP( ip, prefixLen, intf )

# Start network that now includes link to root namespace

network.start()

# Add routes from root ns to hosts

for route in routes:

root.cmd( 'route add -net ' + route + ' dev ' + str( intf ) )

def sshd( network, cmd='/usr/sbin/sshd', opts='-D' ):

"Start a network, connect it to root ns, and run sshd on all hosts."

switch = network.switches[ 0 ] # switch to use

ip = '10.123.123.1' # our IP address on host network

routes = [ '10.0.0.0/8' ] # host networks to route to

connectToRootNS( network, switch, ip, 8, routes )

for host in network.hosts:

host.cmd( cmd + ' ' + opts + '&' )

# DEBUGGING INFO

print("\n")

print("Dumping host connections")

dumpNodeConnections(network.hosts)

print("\n")

print("\*\*\* Hosts are running sshd at the following addresses:")

print("\n")

for host in network.hosts:

print(host.name, host.IP())

print("\n")

print("\*\*\* Type 'exit' or control-D to shut down network")

print("\n")

print("\*\*\* For testing network connectivity among the hosts, wait a bit for the controller to create all the routes, then do 'pingall' on the mininet console.")

print("\n")

CLI( network )

for host in network.hosts:

host.cmd( 'kill %' + cmd )

network.stop()

# by zys

def start\_network(network):

network.start()

# DEBUGGING INFO

print("\n")

print("Dumping host connections")

dumpNodeConnections(network.hosts)

print("\n")

for host in network.hosts:

print(host.name, host.IP())

print("\n")

print("\*\*\* Type 'exit' or control-D to shut down network")

print("\n")

print("\*\*\* For testing network connectivity among the hosts, wait a bit for the controller to create all the routes, then do 'pingall' on the mininet console.")

print("\n")

print("\*\*\* edited for xjtu sdn\_exp\_2020")

print("\n")

CLI( network )

network.stop()

if \_\_name\_\_ == '\_\_main\_\_':

setLogLevel('info')

#setLogLevel('debug')

# sshd( setupNetwork(controller\_ip) )

start\_network(setupNetwork(controller\_ip))

exp3\_1:

from ryu.base import app\_manager

from ryu.ofproto import ofproto\_v1\_3

from ryu.controller.handler import set\_ev\_cls

from ryu.controller.handler import MAIN\_DISPATCHER, CONFIG\_DISPATCHER

from ryu.controller import ofp\_event

from ryu.lib.packet import packet

from ryu.lib.packet import arp

from ryu.lib.packet import ethernet

from ryu.lib.packet import ether\_types

from ryu.lib import hub

from ryu.topology.api import get\_link, get\_switch

import networkx as nx

class NetworkAwareness(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(NetworkAwareness, self).\_\_init\_\_(\*args, \*\*kwargs)

self.dpid\_mac\_port = {}

self.arp\_record = {}

self.topo\_thread = hub.spawn(self.\_get\_topology)

self.network = nx.DiGraph()

@set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)

def packet\_in\_handler(self, ev):

msg = ev.msg

dp = msg.datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

pkt = packet.Packet(msg.data)

eth\_pkt = pkt.get\_protocol(ethernet.ethernet)

dpid = dp.id

in\_port = msg.match['in\_port']

src = eth\_pkt.src

dst = eth\_pkt.dst

if eth\_pkt.ethertype == ether\_types.ETH\_TYPE\_LLDP or eth\_pkt.ethertype== ether\_types.ETH\_TYPE\_IPV6:

return

header\_list = dict( (p.protocol\_name, p) for p in pkt.protocols if type(p) != str )

if src not in self.dpid\_mac\_port:

self.dpid\_mac\_port[src] = (dpid, in\_port)

out\_port = ofp.OFPP\_FLOOD

if dst in self.dpid\_mac\_port:

if dpid == self.dpid\_mac\_port[dst][0]:

self.logger.info('get the final switch')

print ('---------------------------------')

out\_port = self.dpid\_mac\_port[dst][1]

else:

start = self.dpid\_mac\_port[src][0]

end = self.dpid\_mac\_port[dst][0]

path = nx.shortest\_path(self.network, start, end)

if dpid not in path:

return

else:

next\_hop = path[path.index(dpid)+1]

self.logger.info('from src:%s -> dst:%s now:%s -> next:%s, %s', src, dst, dpid, next\_hop, self.network[dpid][next\_hop])

out\_port = self.network[dpid][next\_hop]['port']

else:

ARP = arp.arp.\_\_name\_\_

if dst == "ff:ff:ff:ff:ff:ff" and ARP in header\_list:

arp\_dst\_ip = header\_list[ARP].dst\_ip

if (dpid, src, arp\_dst\_ip) in self.arp\_record:

if self.arp\_record[(dpid, src, arp\_dst\_ip)] != in\_port:

return

else:

self.arp\_record[(dpid, src, arp\_dst\_ip)] = in\_port

actions = [parser.OFPActionOutput(out\_port)]

if out\_port != ofp.OFPP\_FLOOD:

match = parser.OFPMatch(in\_port=in\_port, eth\_dst=dst)

self.add\_flow(dp, 1, match, actions)

data = msg.data if msg.buffer\_id == ofp.OFP\_NO\_BUFFER else None

out = parser.OFPPacketOut(datapath=dp, buffer\_id=msg.buffer\_id,in\_port=in\_port, actions=actions, data=data)

dp.send\_msg(out)

def add\_flow(self, datapath, priority, match, actions):

dp = datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

inst = [parser.OFPInstructionActions(ofp.OFPIT\_APPLY\_ACTIONS,actions)]

mod = parser.OFPFlowMod(datapath=dp, priority=priority,match=match,instructions=inst)

dp.send\_msg(mod)

@set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)

def switch\_features\_handler(self, ev):

msg = ev.msg

dp = msg.datapath

ofp = dp.ofproto

parser = dp.ofproto\_parser

match = parser.OFPMatch()

actions = [parser.OFPActionOutput(ofp.OFPP\_CONTROLLER,ofp.OFPCML\_NO\_BUFFER)]

self.add\_flow(dp, 0, match, actions)

def \_get\_topology(self):

while True:

switch\_list = get\_switch(self, None)

switches = [switch.dp.id for switch in switch\_list]

self.network.add\_nodes\_from(switches)

# get links

links\_list = get\_link(self, None)

links = [(link.src.dpid, link.dst.dpid, {'port':link.src.port\_no}) for link in links\_list]

self.network.add\_edges\_from(links)

# get reverse links

links = [(link.dst.dpid, link.src.dpid, {'port':link.dst.port\_no}) for link in links\_list]

self.network.add\_edges\_from(links)

hub.sleep(.1)

exp3\_2:

from ryu.base import app\_manager

from ryu.controller import ofp\_event

from ryu.controller.handler import CONFIG\_DISPATCHER, MAIN\_DISPATCHER

from ryu.controller.handler import set\_ev\_cls

from ryu.ofproto import ofproto\_v1\_3

from ryu.lib.packet import packet

from ryu.lib.packet import ethernet

from ryu.lib.packet import tcp

from ryu.lib.packet import ether\_types

from ryu.lib.packet import arp

import networkx as nx

from ryu.topology.api import get\_switch,get\_link

from ryu.topology import event

from ryu.base.app\_manager import lookup\_service\_brick

import time

from ryu.lib import hub

from ryu.controller.handler import MAIN\_DISPATCHER, DEAD\_DISPATCHER,CONFIG\_DISPATCHER

from ryu.topology.switches import Switches

from ryu.topology.switches import LLDPPacket

class ARP\_PROXY\_13(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(ARP\_PROXY\_13, self).\_\_init\_\_(\*args, \*\*kwargs)

self.mac\_to\_port = {}

self.network = nx.DiGraph()

self.graph = nx.DiGraph()

self.paths = {}

self.topology\_api\_app=self

self.echo\_latency={}

self.request\_latency={}

self.datapaths={}

self.sw\_module = lookup\_service\_brick('switches')

self.awareness = lookup\_service\_brick('awareness')

self.network\_aware = lookup\_service\_brick('network\_aware')

@set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)

def switch\_features\_handler(self, ev):

datapath = ev.msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

match = parser.OFPMatch()

actions = [parser.OFPActionOutput(ofproto.OFPP\_CONTROLLER, ofproto.OFPCML\_NO\_BUFFER)]

self.add\_flow(datapath, 0, match, actions)

def add\_flow(self, datapath, priority, match, actions, buffer\_id=None):

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT\_APPLY\_ACTIONS, actions)]

if buffer\_id:

mod = parser.OFPFlowMod(datapath=datapath, buffer\_id=buffer\_id, priority=priority, match=match, instructions=inst)

else:

mod = parser.OFPFlowMod(datapath=datapath, priority=priority, match=match, instructions=inst)

datapath.send\_msg(mod)

def mac\_learning(self, datapath, src, in\_port):

self.mac\_to\_port.setdefault((datapath,datapath.id), {})

# learn a mac address to avoid FLOOD next time.

if src in self.mac\_to\_port[(datapath,datapath.id)]:

if in\_port != self.mac\_to\_port[(datapath,datapath.id)][src]:

return False

else:

self.mac\_to\_port[(datapath,datapath.id)][src] = in\_port

return True

@set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)

def \_packet\_in\_handler(self, ev):

msg = ev.msg

datapath = msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

in\_port = msg.match['in\_port']

pkt = packet.Packet(msg.data)

eth = pkt.get\_protocols(ethernet.ethernet)[0]

dst = eth.dst

src = eth.src

dpid=datapath.id

self.mac\_learning(datapath, src, in\_port)

if eth.ethertype == ether\_types.ETH\_TYPE\_LLDP:

try:

src\_dpid, src\_port\_no = LLDPPacket.lldp\_parse(msg.data)

dpid = datapath.id

if self.sw\_module is None:

self.sw\_module = lookup\_service\_brick('switches')

if src\_dpid not in self.paths.keys():

self.paths.setdefault(src\_dpid, {})

for port in self.sw\_module.ports.keys():

if src\_dpid == port.dpid and src\_port\_no == port.port\_no:

port\_data = self.sw\_module.ports[port]

delay = port\_data.delay

self.request\_latency[(src\_dpid,dpid)] = delay

# print('lldp delay between %s and %s is %fms'%(src\_dpid,dpid,delay\*1000))

self.network[src\_dpid][dpid]['weight'] = self.get\_delay(src\_dpid, dpid)

if dpid in self.network:

if dpid not in self.paths[src\_dpid]:

path = nx.shortest\_path(self.network,src\_dpid,dpid)

self.paths[src\_dpid][dpid]=path

path = nx.shortest\_path(self.network,20,25, weight='weight')

print(path)

total = 0

for i in range(len(path)-1):

total += self.get\_delay(path[i], path[i+1])

# print("total:", total)

except Exception as e:

print(e)

print("error occured")

finally:

return

if dst in self.mac\_to\_port[(datapath,datapath.id)]:

out\_port = self.mac\_to\_port[(datapath,datapath.id)][dst]

else:

if self.mac\_learning(datapath, src, in\_port) is False:

out\_port = ofproto.OFPPC\_NO\_RECV

else:

out\_port = ofproto.OFPP\_FLOOD

actions = [parser.OFPActionOutput(out\_port)]

if out\_port != ofproto.OFPP\_FLOOD:

match = parser.OFPMatch(in\_port=in\_port, eth\_dst=dst)

if msg.buffer\_id != ofproto.OFP\_NO\_BUFFER:

self.add\_flow(datapath, 10, match, actions, msg.buffer\_id)

return

else:

self.add\_flow(datapath, 10, match, actions)

data = None

if msg.buffer\_id == ofproto.OFP\_NO\_BUFFER:

data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer\_id=msg.buffer\_id, in\_port=in\_port, actions=actions, data=data)

datapath.send\_msg(out)

def \_send\_echo\_request(self):

for datapath in self.datapaths.values():

parser = datapath.ofproto\_parser

data = "%.6f" % time.time()

data=data.encode('utf-8')

echo\_req = parser.OFPEchoRequest(datapath, data=data)

datapath.send\_msg(echo\_req)

@set\_ev\_cls(ofp\_event.EventOFPEchoReply, MAIN\_DISPATCHER)

def echo\_reply\_handler(self, ev):

try:

latency = time.time() - eval(ev.msg.data)

if ev.msg.datapath.id not in self.echo\_latency.keys():

self.echo\_latency.setdefault(ev.msg.datapath.id,{})

self.echo\_latency[ev.msg.datapath.id] = latency

# print('echo latency %s is %fms'%(ev.msg.datapath.id, latency\*1000))

except:

print("echo reply handler error")

return

def get\_delay(self, src, dst):

try:

fwd\_delay = self.request\_latency[(src,dst)]

re\_delay = self.request\_latency[(dst,src)]

src\_latency = self.echo\_latency[src]

dst\_latency = self.echo\_latency[dst]

delay = (fwd\_delay + re\_delay - src\_latency - dst\_latency)\*(1000/2)

print('the delay between %s and %s dealy is %fms'%(src,dst,delay))

return max(delay, 0)

except:

print("get delay error")

return float('inf')

@set\_ev\_cls(event.EventSwitchEnter,[CONFIG\_DISPATCHER,MAIN\_DISPATCHER])

def get\_topology(self,ev):

#store nodes info into the Graph

switch\_list = get\_switch(self.topology\_api\_app,None) #------------need to get info,by debug

switches = [switch.dp.id for switch in switch\_list]

self.network.add\_nodes\_from(switches)

#store links info into the Graph

link\_list = get\_link(self.topology\_api\_app,None)

#port\_no, in\_port ---------------need to debug, get diffirent from both

links = [(link.src.dpid,link.dst.dpid,{'attr\_dict':{'port':link.dst.port\_no}}) for link in link\_list] #add edge, need src,dst,weigtht

self.network.add\_edges\_from(links)

links = [(link.dst.dpid,link.src.dpid,{'attr\_dict':{'port':link.dst.port\_no}}) for link in link\_list]

self.network.add\_edges\_from(links)

self.\_send\_echo\_request()

@set\_ev\_cls(ofp\_event.EventOFPStateChange,[MAIN\_DISPATCHER, DEAD\_DISPATCHER])

def \_state\_change\_handler(self, ev):

datapath = ev.datapath

if ev.state == MAIN\_DISPATCHER:

if not datapath.id in self.datapaths:

self.logger.debug('Register datapath: %016x', datapath.id)

self.datapaths[datapath.id] = datapath

elif ev.state == DEAD\_DISPATCHER:

if datapath.id in self.datapaths:

self.logger.debug('Unregister datapath: %016x', datapath.id)

del self.datapaths[datapath.id]